Перегрузка функций

В программировании то и дело случается писать функции для схожих действий, выполняемых над различными типами и наборами данных. Возьмите, например, функцию, которая должна возвращать квадрат своего аргумента. В C/C++ возведение в квадрат целого и числа с плавающей точкой – существенно разные операции. Вообще говоря, придется написать две функции – одну, принимающую целый аргумент и возвращающую целое, и вторую, принимающую тип double и возвращающую также double. В C функции должны иметь уникальные имена. Таким образом, перед программистом встает задача придумывать массу имен для различных функций, выполняющих аналогичные действия. Например, SquareInt() и SquareDbl() .

В C++ допускаются перегруженные имена функций (термин взят из лингвистики), когда функции с одним именем можно, тем не менее, идентифицировать по их списку параметров, т.е. контексту, в котором имя употребляется.

Рассмотрим следующий пример с возведением переменных разного типа в квадрат. В нем предусмотрено еще "возведение в квадрат” строки, когда результатом функции должна быть строка, в которой любые символы, кроме пробелов, удваиваются.

```
int Square(int arg)
{
return arg*arg;
}
double Square(double arg)
{
return arg*arg;
}
char *Square(const char *arg, int n)
{
static char res[256];
int j = 0;
while (*arg && j < n)
{ if (*arg != ' ') res[j++] = *arg;
res[j++] = *arg++;}
res[j] = 0;
return res;
}
int main(void)
{
...
int x = 11;
double y = 3.1416;
char msg[] = "Output from overloaded Function!";
printf("Output: %d, %f, %s\n", Square (x) , Square (y) , Square (msg, 32) ) ;
...
}
```

При вызове перегруженной функции компилятор определяет, какую именно функцию требуется вызвать, по типу фактических параметров. Этот процесс называется разрешением перегрузки (перевод английского слова resolution в смысле «уточнение»). Тип возвращаемого функцией значения в разрешении не участвует. Механизм разрешения сводится к тому, чтобы использовать функцию с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта функции, определяющей наибольшее значение:

```
// Возвращает наибольшее из двух целых:
int max(int, int):
// Возвращает подстроку наибольшей длины:
char* max(char*, char*);
// Возвращает наибольшее из первого параметра и длины второго:
int max (int, char*):
```

```
// Возвращает наибольшее из второго параметра и длины первого:
int max (char*, int);
void f(int a, int b, char* c, char* d)
{
cout << max (a, b) << max(c, d) << max(a, c) << max(c, b);
}
```

При вызове функции max компилятор выбирает соответствующий типу фактических параметров вариант функции (в приведенном примере будут последовательно вызваны все четыре варианта функции).

Если точного соответствия не найдено, выполняются продвижения порядковых типов в соответствии с общими правилами преобразования типов, например, bool и char в int, float в double и т. п. Далее выполняются стандартные преобразования типов, например, int в double или указателей в void*. Следующим шагом является выполнение преобразований типа, заданных пользователем, а также поиск соответствий за счет переменного числа аргументов функций. Если соответствие на одном и том же этапе может быть получено более чем одним способом, вызов считается неоднозначным и выдается сообщение об ошибке.

Неоднозначность может появиться при:

- преобразовании типа;
- использовании параметров-ссылок;
- использовании аргументов по умолчанию.

Пример неоднозначности при преобразовании типа:

```
float f(float i)
{
cout << "function float f(float i)" << endl;
return i;
}
double f(double i)
{
cout << "function double f(double i)" << endl;
return i*2;
}
int main()
{
float x = 10.09:
double y = 10.09;
cout << f(x) << endl; // Вызывается f(float)
cout << f(y) << endl; // Вызывается f(double)
cout << f(10)<< endl; // Неоднозначность - как преобразовать 10: во float или double?
...
}
```

Для устранения этой неоднозначности требуется явное приведение типа для константы 10.

Пример неоднозначности при использовании параметров-ссылок: если одна из перегружаемых функций объявлена как int f(int a, int b), а другая – как int f (int a, int &b), то компилятор не сможет узнать, какая из этих функций вызывается, так как нет синтаксических различий между вызовом функции, которая получает параметр по значению, и вызовом функции, которая получает параметр по ссылке.

Пример неоднозначности при использовании аргументов по умолчанию:

```
int f(int a){return a;}
int f(int a, int b = l){return a * b;}
```

```
int main()
{
…
cout << f(10, 2);       // Вызывается вторая функция f(int, int)
cout << f(10); // Неоднозначность - что вызывается:    f(int, int) или f(int) ?
…
}
```

Правила описания перегруженных функций.

- Перегруженные функции должны находиться в одной области видимости, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.
- Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.
- Функции не могут быть перегружены, если описание их параметров отличается только модификаторами const, volatile или использованием ссылки (например, int и const int или int и int&).
- Нельзя перегружать функции, отличающиеся только типом возвращаемого значения.

Перегруженные функции являются, по сути, совершенно различными функциями, идентифицируемыми не только своим именем (оно у них одно и то же), но и списком параметров. Компилятор выполняет т. н. декорирование имен. дополняя имя функции кодовой последовательностью символов, кодирующей тип ее параметров. Тем самым формируется уникальное внутреннее имя.

Несколько примеров того, как для различных прототипов производится декорирование имени функции:

```
void Func(void);
void Func(int);
void Func(int, int);
void Func(*char);
void Func(unsigned);
void Func(const char*);
```

Тип возвращаемого значения никак не отражается на декорировании имени.

Имена функций могут быть перегружены в пределах одной области видимости. Компилятор отличает одну функцию от другой по сигнатуре. Сигнатура задается числом, порядком следования и типами ее параметров.

Проблемы, возникающие при перегрузке функций

В C++ довольно много неявных преобразований типа. Это в определенных ситуациях может привести к проблемам, в том числе создавать неоднозначность при разрешении перегрузки. Но тем не менее при разрешении перегрузки типы, преобразующиеся в друг друга с помощью неявных преобразований, различаются. Общее правило такое: вариант, не требующий преобразований, имеет приоритет.

Одним из проблемных типов является bool. Для совместимости с C существует неявное преобразование bool в int и неявное преобразование любого числового типа и указателя в bool. Это может породить много трудно обнаруживаемых ошибок. Но в простых случаях при разрешении перегрузки bool четко отделяется от int.

```
void FF(int x);
void FF(bool x); // ...
 int x = 6,
y = 5;
```

```
FF(x == y); // FF(bool)
FF(x = y);  // FF(int)
```

Определенные проблемы также доставляют перечисления, тип перечисления неявно преобразуется в целочисленные типы. При перегрузке тип перечисления может четко отделятся от int.

```
enum Qq { One = 1, Two };
void FF(int x);
void FF(Qq x); // ...
FF(One); // FF(Qq)
FF(42);  // FF(int)
```

Семантически и побитово совпадающие типы, например, int и long также различаются при разрешении перегрузки.

```
void FF(int x);
void FF(long x); // ...
FF(42);  // FF(int)
FF(42L); // FF(long)
```

Перегрузка функций членов классов

Все выше сказанное относится и функциям членам классов, только при вызове данных функций необходимо дополнительно указывать либо имя объекта класса, к которому они принадлежат, либо адрес данного объекта

```
class A{
public:
void FF(int x);
void FF(long x);} X,*pX;
main()
{
X.FF(42); X.FF(42L);
pX=new A;
pX->FF(42); pX->FF(42L);
}
```

Конструктор класса является функцией, только специфического вида, то его также можно перегружать, в соответствии с указанными выше правилами, что позволит создавать объекты одного и того же класса, но отличающиеся в момент создания и конструктор которых будет выполнять действия, отличные от действий других конструкторов.